

华中科技大学

课程实验报告

课程名称： 大数据处理

专业班级： 大数据 2101 班

学 号： U202115652

姓 名： 李嘉鹏

指导教师： 石宣化

报告日期： 2024 年 6 月 3 日

计算机科学与技术学院

目录

实验 1: MapReduce 实验-PageRank.....	3
实验 2: Spark 实验-基于命令行、Scala 和 Spark Streaming 实现 WordCount 程序.....	11
实验 3: 图计算实验-基于 Spark GraphX 实现的 DFS 和 SCC 算法	19
实验总结	25

实验 1: MapReduce 实验-PageRank

1.1 实验概述

本实验旨在通过编写和执行基于 MapReduce 编程模型的 PageRank 程序，帮助深入理解 MapReduce 的工作原理，并学会使用 Hadoop 框架进行大规模数据处理。通过此实验能够掌握 MapReduce 编程的基本概念、编写简单的 MapReduce 程序以及将它们运行在分布式环境中。

1.2 实验内容

实验的主要内容是在开源系统 Hadoop 上实现 PageRank 算法，进一步理解 Map & Reduce 原理。

PageRank 算法是搜索引擎不断发展的产物，其核心思想是从许多优质的网页链接过来的网页必定还是优质网页。为了区分网页之间的优劣，PageRank 引入了一个值来评估一个网页的受欢迎程度，也就是 PR 值。PR 值越高说明该网页受欢迎程度越高。

算法开始设定所有网页为同一 PR 值，如果网页总数为 N，则初始 PR 值一般都设置为 1/N。之后通过如下公式对所有网页的 PR 值进行迭代计算。

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

其中，N 表示网页总数，d 是阻尼因子，通常设为 0.85，PR(pi)表示网页 pi 的 PR 值，L(pi)表示网页 pi 链出网页的数目，在图论里成为出度。在有限次迭代后，所有网页的 PR 值会收敛到一个固定的值。当两次迭代之间 PR 值的改变量小于一个设定的阈值时，算法结束。

1.2.1 阶段 1 基础准备

1. 任务描述

配置实验要求的环境（Hadoop 框架），下载实验数据集。

2. 实验设计

依据实验任务书第一部分“Hadoop 安装与配置实验”完成即可。

3. 实验过程

参照厦大大数据实验指导博客“Hadoop3.1.3 安装教程_单机/伪分布式配置_Hadoop3.1.3/Ubuntu18.04(16.04)”，分别完成以下六步：

- ①安装 Linux 系统；
- ②在 Linux 系统中配置 Hadoop 基础环境；
- ③安装 Java 环境；
- ④安装 Hadoop；
- ⑤完成 Hadoop 的单机配置；
- ⑥完成 Hadoop 的伪分布式配置。

最终需要利用指令验证 Hadoop 是否安装成功。

4. 实验结果

按照上述步骤完成 Hadoop 的安装，使用 jps 命令验证发现启动成功，如图 1 所示。

```
hadoop@hadoop-virtual-machine:/usr/local/hadoop$ ./sbin/start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [hadoop-virtual-machine]
2024-06-06 01:46:11,663 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
hadoop@hadoop-virtual-machine:/usr/local/hadoop$ jps
3318 SecondaryNameNode
3446 Jps
2920 NameNode
3081 DataNode
hadoop@hadoop-virtual-machine:/usr/local/hadoop$
```

图 1: Hadoop 安装成功验证

1.2.2 阶段 2 迭代完成 PageRank 值计算

1. 任务描述

根据实验提供的数据集 SNAP-Stanford（含有 281903 个顶点和 2312497 条边），迭代计算各个顶点的 PageRank 值，并输出结果。

2. 实验设计

- ①首先分析数据集格式并进行读入；
- ②遍历数据集，找到所有唯一的顶点，并将其 PageRank 值初始化。由于全局 PageRank 值之和为 1，因此各顶点 PageRank 的初始值为 $1/281903=3.547\times 10^{-6}$ ，

如下所示：

```
public static final float initialPageRank = 1f / 281903f;
```

③利用 MapReduce 进行计算。MapReduce 的原理如图 2 所示。

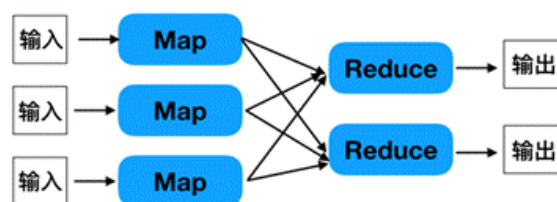


图 2: MapReduce 原理图

在本任务中，具体分为 a~f 共 6 个步骤。

- 读取边，输出(fromNode, toNode)键值对；
- 从前一个任务中读取 PageRank 分数，并输出(fromNode, PageRank 值)键值对；
- 对于每个 fromNode，计算其 toNode 的数量；然后对于每个 toNode，输出(toNode, PageRank 值/数量)键值对。这里完成矩阵乘法步骤中的乘法计算；
- 从 c 的输出中读取数据，并乘以 $1-\beta$ 用于转移；
- 读取初始 PageRank 值或前一轮的 PageRank 值，并乘以 β 用于转移；
- 对于每个 toNode，将所有中间 PR 值相加并输出(toNode, PageRank 值)键值对。

在算法实现中，Map 阶段主要负责数据的转换和计算，而 Reduce 阶段主要负责对 Map 阶段输出的中间结果进行整合和计算最终结果。因此在上述 a~f 六个步骤中，a/b/d/e 属于 Map 阶段，c/f 属于 Reduce 阶段。

在步骤 f 结束后，需要对比全部顶点在前后两轮的 PageRank 值的差异，若其差异全部小于设定好的阈值，那么终止迭代，反之则需要开始新一轮的计算。这部分代码如下所示。

```

// 计算前一轮PageRank值
float previousPR = Float.intBitsToFloat((int) context.getCounter(PREVIOUS_PR).getValue());
// 计算当前PageRank值
float currentPR = 0f;
for (IntWritable value : components) {
    currentPR += previousPR / components.size();
}
// 计算前后两轮PageRank值的差异
float diff = Math.abs(currentPR - state);

// 更新前一轮PageRank值
context.getCounter(PREVIOUS_PR).setValue(Float.floatToIntBits(currentPR));

// 输出当前顶点的PageRank值
context.write(key, new FloatWritable(currentPR));

// 如果差异大于阈值，则继续迭代
if (diff >= threshold)
    context.getCounter(PREVIOUS_PR).setValue(Float.floatToIntBits(Float.MAX_VALUE)); // 重置
    前一轮PageRank值
    context.getCounter(PREVIOUS_PR).increment(1); // 继续迭代

```

详细流程图如图 3 所示。

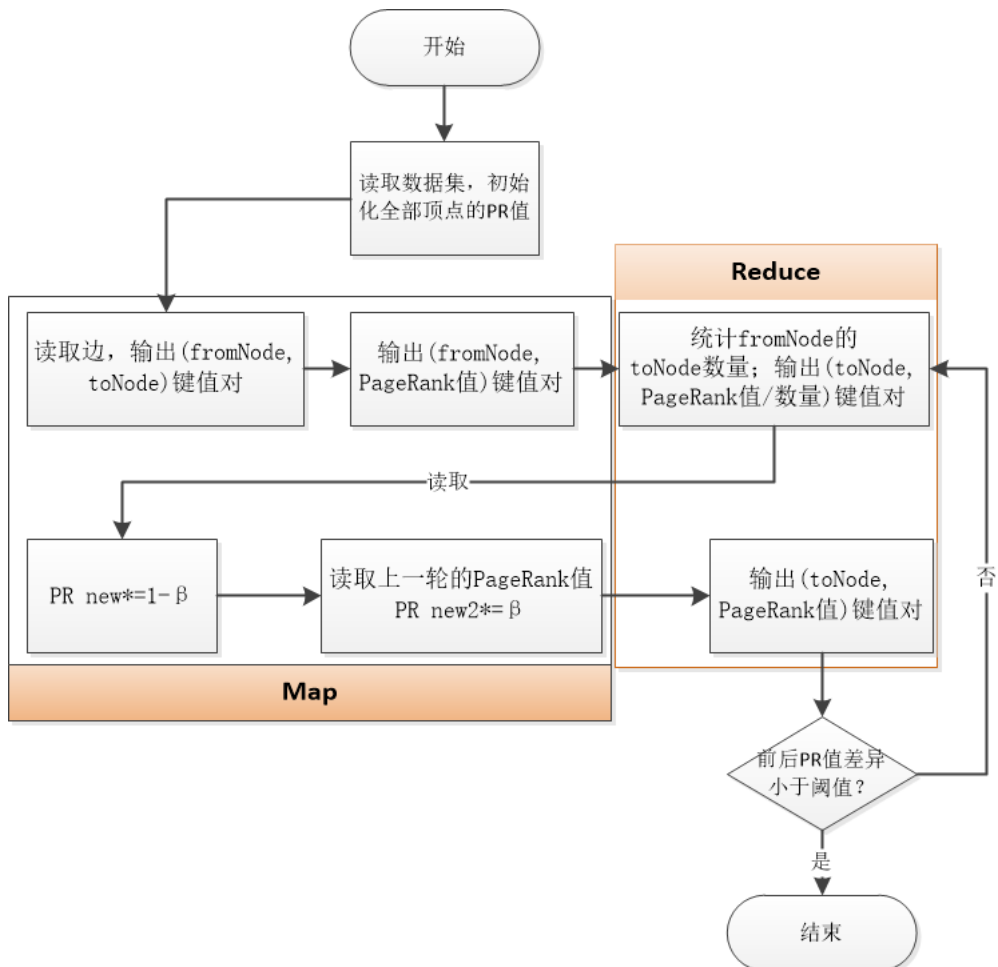


图 3：基于 MapReduce 的 PageRank 算法流程图

3. 实验过程

①读入数据集；

```
DriverInitializer initializer = new DriverInitializer(args);

String[] args0 = {
    initializer.inputFile,
    initializer.initPageRankPath,
    initializer.inputFile
};
ProduceInitPageRankOnHDFS.main(args0);
```

②遍历数据集并将全部顶点的 PageRank 值初始化为 1/281903；

```
public void reduce(IntWritable key, Iterable<NullWritable> values, Context
context) throws IOException, InterruptedException{
    context.write(key, new FloatWritable(HadoopParams.initialPageRank));
}

public static void main(String[] args) throws Exception{
    String inputPath = args[0];
    String pageRankVecPath = args[1];

    // PageRank Vector
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf);
    job.setJarByClass(ProduceInitPageRankOnHDFS.class);

    job.setMapperClass(PageRankVecMapper.class);
    job.setCombinerClass(PageRankVecCombiner.class);
    job.setReducerClass(PageRankVecReducer.class);
```

③a. 读取边，输出(fromNode, toNode)键值对，使用 TransitionMatrixMapper 类完成。由于数据集每行的格式为“FromNodeId ToNodeId”，因此只需要将输入的数据按照分隔符进行分割后存储在数组 fromTo 中，然后将分割后的数组中的第一个和第二个元素作为键值对的键和值输出，并提供给后续的 Map 阶段：

```
public static class TransitionMatrixMapper extends Mapper<Object, Text, IntWritable,
IntWritable>{
    @Override
    public void map(Object key, Text value, Context context) throws
IOException, InterruptedException{
        String[] fromTo = value.toString().trim().split(HadoopParams.SPARATOR);
        if(value.toString().trim().startsWith(HadoopParams.skipSign) ||
fromTo.length<2 || fromTo[1].trim().equals("")) return;
        try {
            context.write(
                new IntWritable(Integer.parseInt(fromTo[0])),
                new IntWritable(Integer.parseInt(fromTo[1]))
            );
        }catch (NumberFormatException e){
            return;
        }
    }
}
```

b. 从前一个任务中读取 PageRank 分数，并输出(fromNode, PageRank 值)键值对。这一步的输出会为下一步的 Reduce 做准备；

```

    public static class PageRankStateMapper extends Mapper<Object, Text, IntWritable, IntFloatWritable>{
        @Override
        public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException{
            String[] sourceState = value.toString().trim().split(HadoopParams.SPARATOR);
            context.write(
                new IntWritable(Integer.parseInt(sourceState[0])),
                new IntFloatWritable(Float.floatToIntBits(Float.parseFloat(sourceState[1])), true)
            );
        }
    }
}

```

c. 对于每个 fromNode，计算 toNode 的数量；然后对于每个 toNode 输出 (toNode, PageRank 值/toNode 数量)键值对。这一步把某顶点的 PageRank 值均分给它指向的顶点，将其 PageRank 值与其转移矩阵分量做乘法，并输出结果；

```

    public static class MultiplicationReducer extends Reducer<IntWritable, IntFloatWritable,
    IntWritable, FloatWritable>{
        @Override
        public void reduce(IntWritable key, Iterable<IntFloatWritable> values, Context context) throws
        IOException, InterruptedException{
            List<IntWritable> components = new ArrayList<IntWritable>();
            float state = 0f;
            for(IntFloatWritable v:values){
                if(v.getState()){
                    state = Float.intBitsToFloat(v.getVal().get());
                }
                else{
                    components.add(v.getVal());
                }
            }

            float pageNum = components.size();
            for(IntWritable value:components){
                float score = state/pageNum;
                context.write(value, new FloatWritable(score));
            }
        }
    }
}

```

d. 从 c 的输出中读取数据，并乘以 $1-\beta$ 用于转移；

```

    public static class PageRankMapper extends Mapper<Object, Text, IntWritable, FloatWritable>{
        public float beta;
        public void setup(Context context){
            Configuration conf = new Configuration();
            beta = conf.getFloat("beta", 0.2f);
        }
        public void map(Object key, Text value, Context context) throws IOException, InterruptedException{
            String[] pageRank = value.toString().trim().split(HadoopParams.SPARATOR);
            String page = pageRank[0];
            float rank = Float.parseFloat(pageRank[1].trim()) * (1-beta);
            context.write(
                new IntWritable(Integer.parseInt(page)),
                new FloatWritable(rank)
            );
        }
    }
}

```

e. 读取初始 PageRank 值或前一个 PageRank 值，并乘以 β 用于转移；

```

    public static class CompensatoryMapper extends Mapper<Object, Text, IntWritable, FloatWritable>{
        public float beta;
        public void setup(Context context){
            Configuration conf = new Configuration();
            beta = conf.getFloat("beta", 0.2f);
        }
        public void map(Object key, Text value, Context context) throws IOException, InterruptedException{
            String[] pageRank = value.toString().trim().split(HadoopParams.SPARATOR);
            String pre_page = pre_pageRank[0];
            float pre_rank = Float.parseFloat(pre_pageRank[1].trim()) * beta;
            context.write(
                new IntWritable(Integer.parseInt(pre_page)),
                new FloatWritable(pre_rank)
            );
        }
    }
}

```


f. 对于每个 toNode, 将所有中间 PR 值相加并输出(toNode, PageRank 值)键值对;

```
public static class PageRankReducer extends Reducer<IntWritable, FloatWritable, IntWritable, FloatWritable>{  
    public void reduce(IntWritable key, Iterable<FloatWritable> values, Context context) throws  
        IOException, InterruptedException{  
        float sum=0f;  
        for(FloatWritable value:values){  
            sum += value.get();  
        }  
        sum = Float.valueOf(HadoopParams.decimalFormat.format(sum));  
        context.write(key, new FloatWritable(sum));  
    }  
}
```

4. 实验结果

设 $\beta=0.85$, 阈值 $\text{threshold}=10^{-3}$ 。当前后两轮顶点的 PageRank 值之差在阈值范围内时, 程序停止, 此时进入 HDFS 文件系统, 可以观察到历次迭代的中间结果如图 4 所示 (节选)。

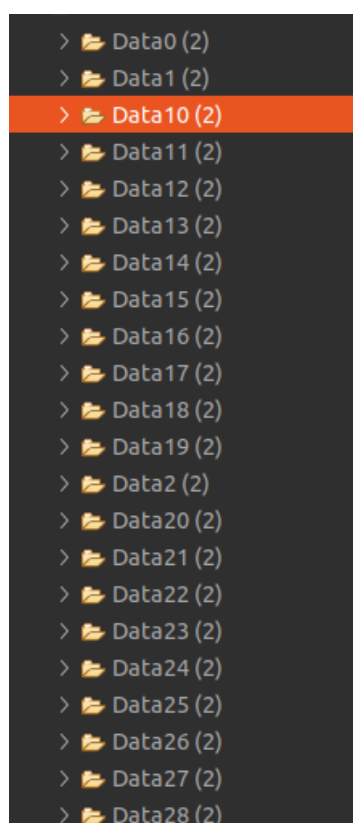





图 4: 迭代过程中间结果 (节选)

最终可以得到全部顶点的 PageRank 值, 如图 5、图 6 所示 (节选, 完整结果见代码文件夹):

Browse Directory

/user/hadoop/PageRank

Go!







Show

25

entries

Search:

<input type="checkbox"/>		Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
<input type="checkbox"/>		-rw-r--r--	hadoop	supergroup	0 B	Jun 06 17:20	1	128 MB	._SUCCESS	
<input type="checkbox"/>		-rw-r--r--	hadoop	supergroup	5.17 MB	Jun 06 17:20	1	128 MB	part-r-00000	

Showing 1 to 2 of 2 entries

Previous

1

Next

图 5: HDFS 中的最终结果文件

1	0.001127661	89073	23	0.0014002685	117152
2	0.009246153	226411	24	0.0013441271	198090
3	0.00827804	241454	25	0.0013063633	60210
4	0.003016195	262860	26	0.0012936015	235496
5	0.0029941653	134832	27	0.0012817546	132695
6	0.0025657215	234704	28	0.001206905	161890
7	0.002447868	136821	29	0.0012051367	112742
8	0.0024249996	68889	30	0.00119105	145892
9	0.0023854484	105607	31	0.0011871683	151428
10	0.0023584052	69358	32	0.0011792216	81435
11	0.0022955264	67756	33	0.001175554	60440
12	0.0022620668	225872	34	0.0011701804	208542
13	0.0022271783	186750	35	0.0011688287	91
14	0.002141434	95163	36	0.001165882	214128
15	0.002141434	272442	37	0.0011651809	258348
16	0.002141434	251796	38	0.0011506351	222873
17	0.002031396	119479	39	0.0011407555	27904
18	0.002002927	231363	40	0.0011407428	272762
19	0.0019525852	55788	41	0.0011335906	93778
20	0.0019117164	167295	42	0.0011297631	96358
21	0.0018060324	179645	43	0.0011052792	181701
22	0.0017343246	38342	44	0.0010829822	259455

图 6: 全部顶点的 PageRank 值最终结果（节选）

1.3 实验小结

本次实验主要是基于 MapReduce 实现 PageRank 算法。PageRank 是谷歌早期用于网页排名的算法之一，在 MapReduce 框架下，PageRank 算法可以被划分为多个任务，每个任务负责不同的计算步骤。在实验中，我通过顶点之间的链接关系来迭代计算每个顶点的 PR 值，主要包括：（1）利用 Mapper 任务读取输入数据，并根据需要进行预处理；（2）利用 Reducer 任务对 Mapper 输出的键值对进行合并操作，迭代计算各顶点的 PR 值；（3）通过 Hadoop 集群管理工具将 MapReduce 任务提交到集群中执行，并监控任务的执行情况。

此次实验的亮点在于我学习到了 MapReduce 框架的常用操作方式，包括 Mapper 和 Reducer 任务的编写和配置，以及如何在 Hadoop 集群上部署和运行 MapReduce 任务代码。通过实验，我进一步掌握了在大数据环境下处理复杂计算任务的技能，深刻体会到了分布式计算的优势在于能够充分利用集群资源来加速计算过程，提高处理大规模数据的效率。

实验 2: Spark 实验-基于命令行、Scala 和 Spark Streaming 实现 WordCount 程序

2.1 实验概述

本实验旨在通过编写和执行基于 Spark 和 Spark Streaming 编程模型的 WordCount 程序,帮助我深入理解 Spark 和 Spark Streaming 的工作原理,并学会使用 Spark 框架进行大规模数据处理。通过此实验,学生将能够掌握 Spark 编程的基本概念、编写简单的 Spark 程序以及将它们在分布式环境中运行。

2.2 实验内容

(1) 基于 Scala 程序实现 WordCount,掌握 RDD 和函数式编程思想。具体包括:

- ①使用命令行执行 WordCount 程序。
- ②使用 Eclipse 编译、打包 WordCount 程序。
- ③查看程序执行结果。

(2) 基于 Spark Streaming 编程模型实现 wordcount 程序。编写 Spark Streaming 程序的基本步骤是:

- ①创建 SparkSession 实例;
- ②创建 DataFrame 表示从数据源输入的每一行数据;
- ③DataFrame 转换,类似于 RDD 转换操作;
- ④创建 StreamingQuery 开启流查询;
- ⑤调用 StreamingQuery.awaitTermination()方法,等待流查询结束。

2.2.1 阶段 1 使用命令行执行 WordCount 程序

1. 任务描述

使用命令行来实现 wordcount,即统计文件中各个单词的计数。在 spark-shell 中读取 Linux 系统本地文件 <file:///usr/local/hadoop/README.txt>,并以空格作为分隔符统计各单词出现的次数。

2. 实验设计

根据实验任务书,使用命令行执行 scala 代码需要在命令行中输入下列语句。

```
scala> val sc = new SparkContext( )
```

```
scala> val textFile = sc.textFile("file:///usr/local/hadoop/README.txt")

// 注：文件路径与任务书不同

scala> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word
=> (word,1)).reduceByKey((a,b) => a+b)

scala> wordCount.collect()

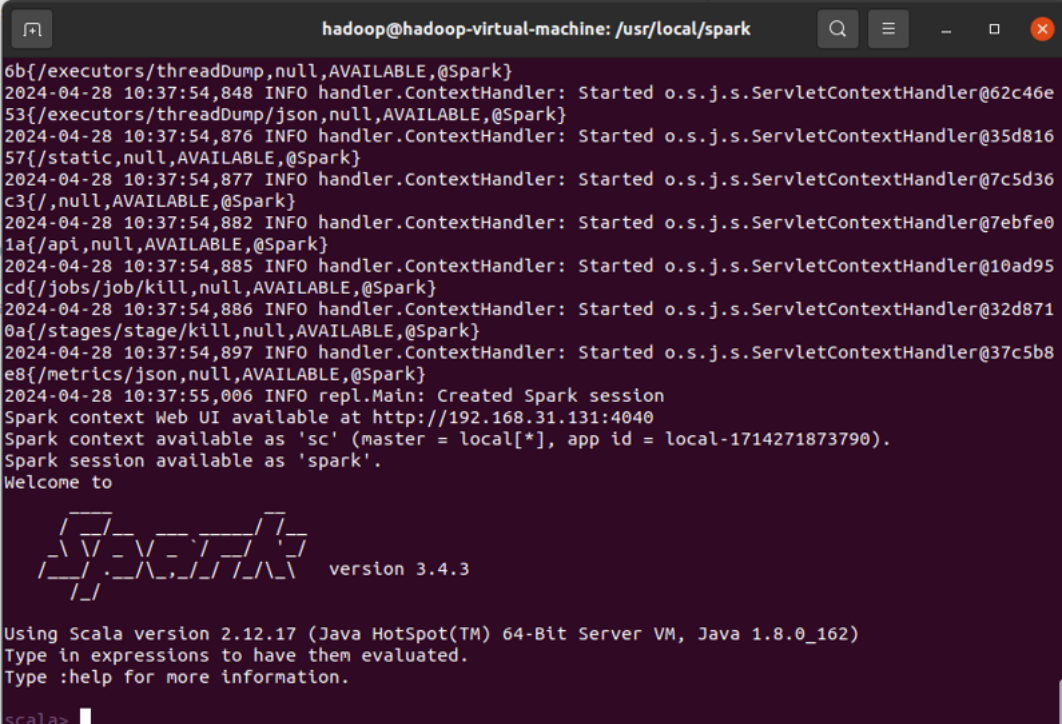
scala> wordCount.foreach(println)
```

3. 实验过程

- ①输入命令 `cd /usr/local/spark`, `bin/spark-shell` 启动 spark;
- ②在命令行中输入实验设计部分的语句。

4. 实验结果

- ①启动 spark, 如图 7 所示;



```
hadoop@hadoop-virtual-machine: /usr/local/spark
6b{/executors/threadDump,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,848 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@62c46e
53{/executors/threadDump/json,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,876 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@35d816
57{/static,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,877 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@7c5d36
c3{/,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,882 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@7ebfe0
1a{/api,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,885 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@10ad95
cd{/jobs/job/kill,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,886 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@32d871
0a{/stages/stage/kill,null,AVAILABLE,@Spark}
2024-04-28 10:37:54,897 INFO handler.ContextHandler: Started o.s.j.s.ServletContextHandler@37c5b8
e8{/metrics/json,null,AVAILABLE,@Spark}
2024-04-28 10:37:55,006 INFO repl.Main: Created Spark session
Spark context Web UI available at http://192.168.31.131:4040
Spark context available as 'sc' (master = local[*], app id = local-1714271873790).
Spark session available as 'spark'.
Welcome to

      _/ _ \| | | | _/_/
     / _ \| | | | |/_/
    / ___| | | | |/_/
   /___|_|_|_|_|/_/

version 3.4.3

Using Scala version 2.12.17 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_162)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

图 7: 成功启动 Spark

- ②使用命令行实现 `wordcount`, 可以看到统计出了指定文件中各个单词的出现次数, 并显示在命令行中, 如图 8 所示。

```
hadoop@hadoop-virtual-machine: /usr/local/spark
2024-04-28 13:12:52,747 INFO storage.ShuffleBlockFetcherIterator: Getting 2 (702.0 B) non-empty b
locks including 2 (702.0 B) local and 0 (0.0 B) host-local and 0 (0.0 B) push-merged-local and 0
(0.0 B) remote blocks
2024-04-28 13:12:52,750 INFO storage.ShuffleBlockFetcherIterator: Started 0 remote fetches in 4 m
s
(under,1)
(this,3)
(distribution,2)
(Technology,1)
(country,1)
(is,1)
(Jetty,1)
(currently,1)
(permitted.,1)
(check,1)
(have,1)
(Security,1)
(U.S.,1)
(with,1)
(BIS,1)
(This,1)
(mortbay.org.,1)
((ECCN),1)
(using,2)
(security,1)
(Department,1)
(export,1)
(reside,1)
(any,1)
(algorithms.,1)
(from,1)
(re-export,2)
(has,1)
(SSL,1)
(Industry,1)
(Administration,1)
(details,1)
(provides,1)
(http://hadoop.apache.org/core/,1)
(country's,1)
(Unrestricted,1)
(740.13),1)
(policies,1)
(country,,1)
```

图 8: WordCount 程序输出结果（命令行）

2.2.2 阶段 2 使用 Eclipse 编译、打包 WordCount 程序

1. 任务描述

用 scale 语言实现 wordcount 程序，需要使用编译和打包的方式运行。

2. 实验设计

根据实验任务书，WordCount 程序的代码如下。

```
1. import org.apache.spark.rdd.RDD
2. import org.apache.spark.{SparkConf, SparkContext}
3.
4. /** * Scala 原生实现 wordcount */
5. object WordCount {
6.     def main(args: Array[String]): Unit = {
7.
```

```

8.      val list = List("cw is cool", "wc is beautiful", "andy is beautiful", "mike is cool")
9.      /** * 第一步，将list 中的元素按照分隔符这里是空格拆分，然后展开 * 先
      map(_.split(" "))将每一个元素按照空格拆分 * 然后flatten 展开 * flatmap 即为
      上面两个步骤的整合 */
10.     val res0 = list.map(_.split(" ")).flatten
11.     val res1 = list.flatMap(_.split(" "))
12.     println("第一步结果")
13.     println(res0)
14.     println(res1)
15.
16.     /** * 第二步是将拆分后得到的每个单词生成一个元组 * k 是单词名称，v 任意
      字符即可这里是1 */
17.     val res3 = res1.map((_, 1))
18.     println("第二步结果")
19.     println(res3)
20.     /** * 第三步是根据相同的key 合并 */
21.     val res4 = res3.groupBy(_._1)
22.     println("第三步结果")
23.     println(res4)
24.     /** * 最后一步是求出groupBy 后的每个key 对应的value 的size 大小，即单
      词出现的个数 */
25.     val res5 = res4.mapValues(_.size)
26.     println("最后一步结果")
27.     println(res5.toBuffer)
28.   }
29. }

```

3. 实验过程

编译并打包上面的程序需要使用到 sbt，具体安装过程参照厦大数据实验指导博客“大数据原理与应用 第十六章 Spark 学习指南”。安装成功后，在命令行输入下面的指令完成打包：

```

1.   cd ~           # 进入用户主文件夹
2.   mkdir ./sparkapp      # 创建应用程序根目录
3.   mkdir -p ./sparkapp/src/main/scala      # 创建所需的文件夹结构
4.   /usr/local/sbt/sbt package

```

最后，通过 spark-submit 运行程序：

```

1.   /usr/local/spark/bin/spark-submit --class "Wordcount" ~/sparkapp/target/scala-2.12/simple-project_2.12-1.0.jar

```


4. 实验结果

①打包成功的结果如图 9 所示；

```
[info] Done updating.
[info] Compiling 3 Scala sources to /home/hadoop/sparkapp/target/scala-2.12/classes...
[warn] Multiple main classes detected. Run 'show discoveredMainClasses' to see the list
[info] Packaging /home/hadoop/sparkapp/target/scala-2.12/simple-project_2.12-1.0.jar ...
[info] Done packaging.
[success] Total time: 10 s, completed 2024-6-6 21:02:09
```

图 9: WordCount 程序打包成功

②最终结果如图 10 所示。

```
第一步结果
List(cw, is, cool, wc, is, beautiful, andy, is, beautiful, mike, is, cool)
List(cw, is, cool, wc, is, beautiful, andy, is, beautiful, mike, is, cool)
第二步结果
List((cw,1), (is,1), (cool,1), (wc,1), (is,1), (beautiful,1), (andy,1), (is,1), (beautiful,1), (mike,1), (is,1), (cool,1))
第三步结果
Map(beautiful -> List((beautiful,1), (beautiful,1)), is -> List((is,1), (is,1), (is,1), (is,1)), wc -> List((wc,1)), andy -> List((andy,1)), cool -> List((cool,1), (cool,1)), cw -> List((cw,1)), mike -> List((mike,1)))
最后一步结果
ArrayBuffer((beautiful,2), (is,4), (wc,1), (andy,1), (cool,2), (cw,1), (mike,1))
```

图 10: WordCount 程序输出结果（Scala 代码打包编译）

2.2.3 阶段 3 Spark Streaming 实验

1. 任务描述

基于 Spark Streaming 编程模型实现可处理实时流查询的 WordCount 程序。

2. 实验设计

根据实验任务书完成即可。

①首先打开一个终端，输入以下命令：

1. `cd /usr/local/spark/mycode`
2. `mkdir streaming`
3. `cd streaming`
4. `mkdir -p src/main/scala`

```
5. cd src/main/scala
6. vim TestStructuredStreaming.scala
```

②用 vim 打开一个 scala 文件，在该文件输入以下内容：

```
1. import org.apache.spark.sql.functions._
2. import org.apache.spark.sql.Session
3.
4. object WordCountStructuredStreaming{
5.   def main(args: Array[String]){
6.     val spark = SparkSession.builder.appName("StructuredNetworkWordCount")
       .getOrCreate()
7.     import spark.implicits._
8.     val lines = spark.readStream.format("socket").option("host","localhost")
       .option("port",9999).load()
9.     val words = lines.as[String].flatMap(_.split(" "))
10.    val wordCounts = words.groupBy("value").count()
11.    val query = wordCounts.writeStream.outputMode("complete").format("console")
       .start()
12.    query.awaitTermination()
13.  }
14. }
```

③代码写好后退出终端，然后在/usr/local/spark/mycode/streaming 目录下创建 simple.sbt 文件：

```
1. cd /usr/local/spark/mycode/streaming
2. vim simple.sbt
```

打开 vim 编辑器后输入以下内容：

```
1. name := "Simple Project"
2. version := "1.0"
3. scalaVersion := "2.12.17"
4. libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.0.0"
```

④执行 sbt 打包编译：

```
1. cd /usr/local/spark/mycode/streaming
2. /usr/local/sbt/sbt package
```

打包成功以后，输入以下命令启动程序：

1. `cd /usr/local/spark/mycode/streaming`
2. `/usr/local/spark/bin/spark-submit --class "WordCountStructuredStreaming" ./target/scala-2.12/simple-project_2.12-1.0.jar`

执行以上输出程序后就开启了监听状态，在终端输入“hello world, hello Beijing, hello world”之后，spark 应用终端即可输出“hello”和“world”单词出现的次数。

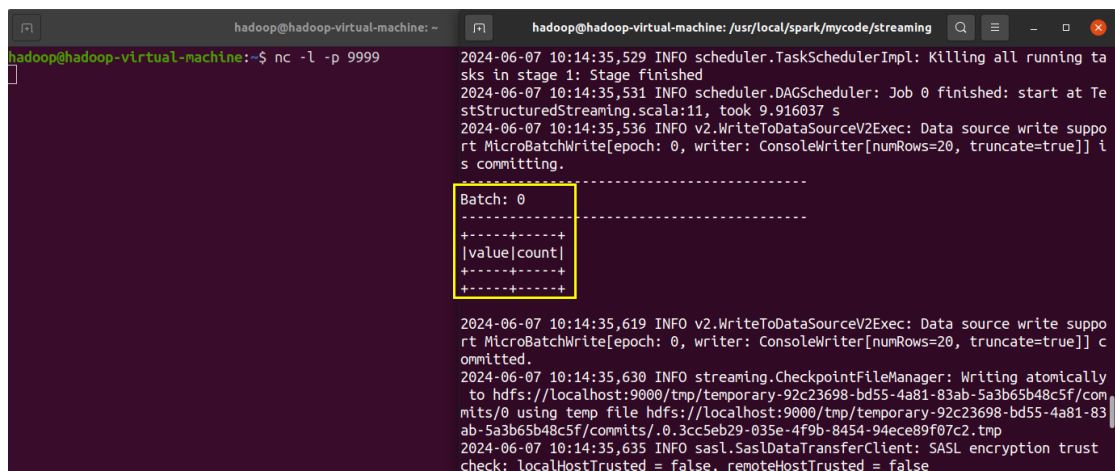
3. 实验过程

按照上述步骤依次执行即可（设当前终端为 A）。当代码打包编译成功后，需要新开另一个终端 B，并输入命令 `nc -l -p 9999`。

此时监听开始，在终端 B 中输入内容后，终端 A 就会实时显示出 B 中句子的 WordCount 统计结果。

4. 实验结果

如图 11~图 13 所示，可以发现最开始 WordCount 为空；一旦监听启动，WordCount 的统计结果会始终记录此前的所有输入，而不是新来一句话就重置一次统计结果，这也反映了流数据处理的特征。



```
hadoop@hadoop-virtual-machine: ~  
hadoop@hadoop-virtual-machine:~$ nc -l -p 9999  
2024-06-07 10:14:35,529 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished  
2024-06-07 10:14:35,531 INFO scheduler.DAGScheduler: Job 0 finished: start at TestStructuredStreaming.scala:11, took 9.916037 s  
2024-06-07 10:14:35,536 INFO v2.WriteToDataSourceV2Exec: Data source write support MicroBatchWrite[epoch: 0, writer: ConsoleWriter[numRows=20, truncate=true]] is committing.  
-----  
Batch: 0  
-----  
+----+-----+  
|value|count|  
+----+-----+  
+----+-----+  
2024-06-07 10:14:35,619 INFO v2.WriteToDataSourceV2Exec: Data source write support MicroBatchWrite[epoch: 0, writer: ConsoleWriter[numRows=20, truncate=true]] committed.  
2024-06-07 10:14:35,630 INFO streaming.CheckpointFileManager: Writing atomically to hdfs://localhost:9000/tmp/temporary-92c23698-bd55-4a81-83ab-5a3b65b48c5f/commits/0 using temp file hdfs://localhost:9000/tmp/temporary-92c23698-bd55-4a81-83ab-5a3b65b48c5f/commits/.0.3cc5eb29-035e-4f9b-8454-94ece89f07c2.tmp  
2024-06-07 10:14:35,635 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
```

图 11: Spark Streaming 结果（1）

```
hadoop@hadoop-virtual-machine: ~  
hadoop@hadoop-virtual-machine:~$ nc -l -p 9999  
hello world, hello Beijing, hello world  
rt MicroBatchWrite[epoch: 1, writer: ConsoleWriter[numRows=20, truncate=true]] i  
s committing.  
Batch: 1  
-----  
2024-06-07 10:17:14,956 INFO codegen.CodeGenerator: Code generated in 8.955866 m  
s  
2024-06-07 10:17:14,975 INFO codegen.CodeGenerator: Code generated in 14.429368  
ms  
-----  
+-----+  
| value|count|  
+-----+  
|world, hello| 1|  
|hello| 2|  
|Beijing,| 1|  
|world| 1|  
+-----+  
2024-06-07 10:17:14,979 INFO v2.WriteToDataSourceV2Exec: Data source write suppo  
rt MicroBatchWrite[epoch: 1, writer: ConsoleWriter[numRows=20, truncate=true]] c  
ommitted.  
2024-06-07 10:17:14,991 INFO streaming.CheckpointFileManager: Writing atomically  
to hdfs://localhost:9000/tmp/temporary-92c23698-bd55-4a81-83ab-5a3b65b48c5f/com  
mits/1 using temp file hdfs://localhost:9000/tmp/temporary-92c23698-bd55-4a81-83
```

图 12: Spark Streaming 结果（2）

```
hadoop@hadoop-virtual-machine: ~  
hadoop@hadoop-virtual-machine:~$ nc -l -p 9999  
hello world, hello Beijing, hello world  
big data experiment, completed by LJP  
2024-06-07 10:18:58,424 INFO v2.WriteToDataSourceV2Exec: Data source write suppo  
rt MicroBatchWrite[epoch: 2, writer: ConsoleWriter[numRows=20, truncate=true]] i  
s committing.  
Batch: 2  
-----  
+-----+  
| value|count|  
+-----+  
|by| 1|  
|completed| 1|  
|world, hello| 1|  
|hello| 2|  
|LJP| 1|  
|data| 1|  
|Beijing,| 1|  
|experiment,| 1|  
|world| 1|  
|big| 1|  
+-----+  
2024-06-07 10:18:58,464 INFO v2.WriteToDataSourceV2Exec: Data source write suppo  
rt MicroBatchWrite[epoch: 2, writer: ConsoleWriter[numRows=20, truncate=true]] c  
ommitted.
```

图 13: Spark Streaming 结果（3）

2.3 实验小结

在本次实验中，我使用 Scala 编写了简单的 WordCount 程序。通过以命令行和打包编译两种方式将其在分布式环境中运行，提高了我的编程能力、调试经验和分布式计算技能。

除此之外，我还对 Spark Streaming 的工作原理有了更深入的理解，包括 RDD 和 DataFrame 的概念以及它们在大数据处理中的应用。与传统的批处理不同，Spark Streaming 允许以微批次的方式处理实时流数据，而通过实现一个面向实时流查询的 WordCount 程序，我更加熟悉了 Spark Streaming 的基本操作，也了解了 Spark Streaming 的容错机制和流式计算模型，这为我的实时数据分析和处理打下了很好的基础。

综上所述，通过本次实验，我加深了对函数式编程思想的理解，并学会了如何将其应用于大规模数据的处理任务中。

实验 3: 图计算实验-基于 Spark GraphX 实现的 DFS 和 SCC 算法

3.1 实验概述

本实验旨在通过编写和执行基于 Spark 的 GraphX 的 DFS（深度优先搜索）和 SCC（强联通分量）程序，帮助学生深入理解图计算系统的工作原理，并学会使用 Spark GraphX 进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写简单的图算法程序以及将它们运行在 GraphX 系统中。

3.2 实验内容

由于基于 YiTu 的图计算实验需要安装 CUDA 编程库，而电脑上没有 GPU，因此需要采用 Spark 的 GraphX 来实现图的深度优先搜索 DFS 和强连通分量 SCC 算法。数据集：<https://github.com/databricks/spark-training/tree/master/data/graphx>。

具体来说，实验需要在给定图数据集上完成深度优先搜索 DFS 和强连通分量 SCC 两种算法的实现，并输出计算结果。下面将会详细介绍两种算法。

3.2.1 阶段 1 深度优先搜索 DFS

1. 任务描述

深度优先搜索（Depth First Search, DFS）是一种用于图的搜索算法，用于遍历或搜索图中的节点。该算法从起始节点开始，沿着图的一条路径一直向前探索，直到到达最深的节点，然后回溯到之前的节点继续探索其它路径，直到所有节点都被访问过。具体步骤是：

- ①选择一个起始节点开始遍历；
- ②访问该节点，并将其标记为“已访问”；
- ③从当前节点开始，选择一个未访问过的相邻节点，将其作为下一个要访问的节点，然后重复步骤②；
- ④如果当前节点没有未访问过的相邻节点，则回溯到上一个节点，继续探索其他路径；

⑤重复步骤③和步骤④，直到所有节点都被访问过。

DFS 的特点是它会尽可能深地搜索图的路径，直到无法再深入为止，然后回溯并继续搜索其他路径。它使用堆栈数据结构来实现，通过堆栈来保存要访问的节点，以便在回溯时能够正确返回上一个节点。但是 DFS 在解决问题时可能会陷入无限循环的情况，尤其是对于包含循环的图。为了避免这种情况，必须使用一种方法来记录已访问的节点。

2. 实验设计

根据上述分析，可以写出 DFS 算法的 java 代码。

①读取输入数据：顶点数据从数据集 graphx-wiki-vertices.txt 中读取，每行包含一个顶点的 id 和名称，以制表符分隔；边数据从数据集 graphx-wiki-edges.txt 中读取，每行包含一条边的源顶点 id 和目标顶点 id，以制表符分隔。然后使用读取的顶点数据和边数据构建图对象 graph。

②定义递归函数 dfs：该函数用于深度优先搜索，接受起始顶点 id 和已访问顶点的集合两个参数。它会递归探索图中的顶点，并返回访问顺序的列表。

③执行深度优先搜索：调用 dfs 函数，得到访问顺序的列表 visitedOrder，最终输出访问顺序。

3. 实验过程

DFS 算法的完整代码如下。

```
1. import org.apache.spark.{SparkConf, SparkContext}
2. import org.apache.spark.graphx._
3.
4. object DFS {
5.   def main(args: Array[String]): Unit = {
6.     val conf = new SparkConf().setAppName("DFSExample").setMaster("local[*]")
7.     val sc = new SparkContext(conf)
8.
9.     // 读取顶点和边数据
10.    val vertices = sc.textFile("graphx-wiki-vertices.txt")
11.    .map { line =>
```

```

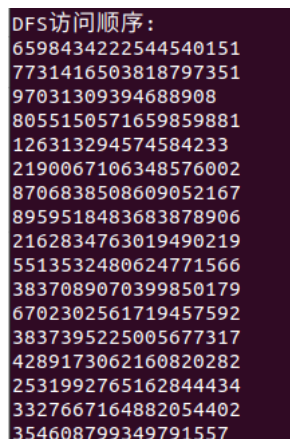
12.         val Array(id, name) = line.split("\t")
13.         (id.toLong, name)
14.     }
15.     val edges = sc.textFile("graphx-wiki-edges.txt")
16.     .map { line =>
17.         val Array(src, dst) = line.split("\t")
18.         Edge(src.toLong, dst.toLong, 1)
19.     }
20.
21.     // 构建 Graph 对象
22.     val graph = Graph(vertices, edges)
23.
24.     // 定义 DFS 函数
25.     def dfs(start: VertexId, visited: Set[VertexId]): List[VertexId]
      = {
26.         if (visited.contains(start)) {
27.             List.empty[VertexId]
28.         } else {
29.             val newVisited = visited + start
30.             val neighbors = graph.edges.filter(_._srcId == start).map(_._dstId).collect()
31.             start :: neighbors.flatMap(neighbor => dfs(neighbor, newVisited)).toList
32.         }
33.     }
34.     // 初始顶点
35.     val startVertex = 6598434222544540151L
36.     // 执行 DFS
37.     val visitedOrder = dfs(startVertex, Set.empty[VertexId])
38.
39.     // 输出访问顺序
40.     println("DFS 访问顺序:")
41.     visitedOrder.foreach(println)
42.     sc.stop()
43. }
44. }

```

该代码的打包编译流程与 2.2.2 小节中所述的完全一致，此处不再赘述。

4. 实验结果

运行上述 DFS 代码，输出结果如图 14 所示（节选）。可以看到输出了顶点的访问先后顺序，且第一个被访问的顶点是代码中预先设定好的初始顶点，说明 DFS 算法是有效的。



```
DFS访问顺序:
6598434222544540151
7731416503818797351
97031309394688908
8055150571659859881
126313294574584233
2190067106348576002
8706838508609052167
8959518483683878906
2162834763019490219
5513532480624771566
3837089070399850179
6702302561719457592
3837395225005677317
4289173062160820282
2531992765162844434
3327667164882054402
354608799349791557
```

图 14: DFS 算法结果（节选）

3.2.2 阶段 2 强联通分量 SCC

1. 任务描述

强连通分量（Strongly Connected Components，SCC）是一种用于在有向图中识别强连通分量的算法。强连通分量是指图中的一组顶点，其中任意两个顶点都可以通过图中的路径相互到达，即两两之间可达。

在 Spark 中，GraphX 框架提供了用于计算图的强连通分量的库，其中的 StronglyConnectedComponents 类实现了用于计算强连通分量的算法。

2. 实验设计

- ①读取输入数据：具体实现与第 3.2.1 节 DFS 算法完全一致，此处不再赘述。
- ②运行 SCC 算法：构建图对象并调用图对象的 stronglyConnectedComponents 方法来运行强连通分量算法，且最多迭代 5 次。
- ③输出结果：从步骤②的结果中提取出顶点及其对应的强连通分量 id，即 scc。将强连通分量按照 id 分组后遍历分组后的强连通分量，最终输出每个强连通分量包含的顶点。

3. 实验过程

SCC 算法的完整代码如下。

```
1.  import org.apache.spark.graphx._
2.  import org.apache.spark.{SparkConf, SparkContext}
3.
4.  object SCC {
5.      def main(args: Array[String]): Unit = {
6.          val conf = new SparkConf().setAppName("SCC").setMaster("local[*]
7.          ")
8.          val sc = new SparkContext(conf)
9.
10.         // 读取顶点和边的数据
11.         val vertices = sc.textFile("graphx-wiki-vertices.txt").map { lin
12.             e =>
13.                 val fields = line.split("\t")
14.                 (fields(0).toLong, fields(1))
15.             }
16.         val edges = sc.textFile("graphx-wiki-edges.txt").map { line =>
17.             val fields = line.split("\t")
18.             Edge(fields(0).toLong, fields(1).toLong, 1)
19.         }
20.
21.         // 构建图
22.         val defaultVertex = (")
23.         val graph = Graph(vertices, edges, defaultVertex)
24.
25.         // 运行强连通分量算法
26.         val scc = graph.stronglyConnectedComponents(5).vertices
27.         // 将结果收集并按照强连通分量id 分组
28.         val sccGroups = scc.map(_._swap).groupByKey().values
29.         // 打印结果
30.         var sccCount = 0
31.         sccGroups.collect().foreach { group =>
32.             sccCount += 1
33.             val vertices = group.mkString(", ")
34.             println(s"强联通分量$sccCount 包含顶点 $vertices")
35.         }
36.         sc.stop()
37.     }
38. }
```

4. 实验结果

运行上述 SCC 代码，输出结果如图 15 所示（节选）。程序输出了每个强联通分量组所包含的顶点元素，位于同组的顶点之间是相互可达的，这也意味着每个强联通分量就是一个子图。例如，下图中前 11 行的顶点均属于强联通分量 1，构成了一个子图。

```
强联通分量1 包含顶点 7529982852778177550
强联通分量1 包含顶点 7712105926825250494
强联通分量1 包含顶点 3240424614053302802
强联通分量1 包含顶点 3837156315242229994
强联通分量1 包含顶点 2881954944517966888
强联通分量1 包含顶点 7079590857970087648
强联通分量1 包含顶点 8270103566577939122
强联通分量1 包含顶点 2826711431471136406
强联通分量1 包含顶点 1049155426026764359
强联通分量1 包含顶点 3506117503017089330
强联通分量1 包含顶点 3345013687889028887
强联通分量2 包含顶点 5601121075172382259
强联通分量2 包含顶点 1544573277905017760
强联通分量2 包含顶点 1735229052138995773
强联通分量2 包含顶点 8264667161902196336
```

图 15: SCC 算法结果（节选）

3.3 实验小结

在本实验中，我学会了如何使用 Spark GraphX 库来构建和处理图数据，包括读取顶点和边的数据、构建图对象、运行图算法等。这为我在大规模图数据分析和处理方面打下了基础，提高了我对图计算理论的理解。

DFS 和 SCC 算法都是应用范围十分广阔的图算法，其中 DFS 算法对于连通性分析、拓扑排序等问题具有重要意义，SCC 算法则有助于进行快速的社交网络分析。在本实验中，它们都是在 Spark 分布式计算框架下运行的，并且均可以高效处理大规模的图数据集，这也让我对分布式计算和并行处理的概念有了更深入的了解。

实验总结

本学期的大数据处理实验结束了，这也是我大学期间最后一次与理论课结合的课程实验。这三次实验让我受益良多，也为本专业的大数据系列课程画上了句号。

在实验一中，我学习了如何使用 MapReduce 框架实现 PageRank 算法。通过将 PageRank 的迭代计算划分为多个任务，并利用 Mapper 和 Reducer 任务对数据进行处理和合并，我深入理解了 MapReduce 的技术原理。我还学会了如何在 Hadoop 集群上部署和运行 MapReduce 任务代码，掌握了 Hadoop 的配置与安装，深入理解了 HDFS 系统的独特优势，这为我处理大规模数据和复杂计算任务奠定了坚实的基础。

在实验二中，我使用 Scala 编写了 WordCount 程序，并在分布式环境中运行。通过掌握命令行和打包编译两种方式，我增强了自己的分布式计算技能，拓宽了个人技术栈。同时，我还对 Spark Streaming 的工作原理有了更深入的理解，通过实现一个实时流查询的 WordCount 程序，熟悉了 Spark Streaming 的常见操作和流式计算模型，增强了我对实时数据分析和处理的信心。

在实验三中，我学习了如何使用 Spark GraphX 框架来构建和处理图数据，包括读取顶点和边数据、构建图对象、根据实际需求对图对象进行具体查询等。这为我在大规模图数据分析和处理方面打下了基础，提高了对图计算理论的理解，对于图数据的挖掘具有重要作用。我还探索了 DFS 和 SCC 算法的实际应用，让我看到了分布式计算在处理大规模图数据时的优势。

综上所述，这三次实验极大地增强了我的动手能力与代码实践经验。在未来，我希望能够进一步探索分布式系统和大数据处理技术，深入研究图计算、实时流处理和机器学习等领域，提高自己在数据科学和人工智能方面的能力。我也希望能够将所学知识和技能应用到实际项目中（尤其是当下前沿的 AI For Science 领域），解决真实世界中的问题，为社会和科技发展做出贡献。

最后，无比感谢课程组老师的辛勤付出！